are solved by back substitution. This yields

$$x_3 = 18/-9 = -2$$
$$x_2 = (6 + 2x_3)/2 = (6 + 2(-2))/2 = 1$$
$$x_1 = 7 - 4x_2 - x_3 = 7 - 4(1) - (-2) = 5$$

EXAMPLE 2.6

Compute Choleski's decomposition of the matrix

$$\mathbf{A} = \begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 11 \end{bmatrix}$$

Solution First, we note that **A** is symmetric. Therefore, Choleski's decomposition is applicable, provided that the matrix is also positive definite. An *a priori* test for positive definiteness is not needed, since the decomposition algorithm contains its own test: if square root of a negative number is encountered, the matrix is not positive definite and the decomposition fails.

Substituting the given matrix for **A** in Eq. (2.16), we obtain

$$\begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 11 \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{bmatrix}$$

Equating the elements in the lower (or upper) triangular portions yields

$$L_{11} = \sqrt{4} = 2$$
$$L_{21} = -2/L_{11} = -2/2 = -1$$
$$L_{31} = 2/L_{11} = 2/2 = 1$$
$$L_{22} = \sqrt{2 - L_{21}^2} = \sqrt{2 - 1^2} = 1$$
$$L_{32} = \frac{-4 - L_{21}L_{31}}{L_{22}} = \frac{-4 - (-1)(1)}{1} = -3$$
$$L_{33} = \sqrt{11 - L_{31}^2 - L_{32}^2} = \sqrt{11 - (1)^2 - (-3)^2} = 1$$

Therefore,

$$\mathbf{L} = \begin{bmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -3 & 1 \end{bmatrix}$$

The result can easily be verified by performing the multiplication $\mathbf{L}\mathbf{L}^T$.

EXAMPLE 2.7

Solve **AX** = **B** with Doolittle's decomposition and compute |**A**|, where

$$\mathbf{A} = \begin{bmatrix} 3 & -1 & 4 \\ -2 & 0 & 5 \\ 7 & 2 & -2 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 6 & -4 \\ 3 & 2 \\ 7 & -5 \end{bmatrix}$$

Solution In the program below, the coefficient matrix **A** is first decomposed by calling **LUdec.** Then **LUsol** is used to compute the solution one vector at a time.

```
% Example 2.7 (Doolittle's decomposition)
A = [3  -1  4; -2  0  5; 7  2  -2];
B = [6  -4; 3  2; 7  -5];
A = LUdec(A);
det = prod(diag(A))
```

```
for i = 1: size(B,2)
        X( : , i) = LUsol(A,B(: , i));
end
X
```

Here are the results:

```
>> det =
        -77
X =
    1.0000   -1.0000
    1.0000    1.0000
    1.0000    0.0000
```

EXAMPLE 2.8

Solve the equations **Ax** = **b** by Choleski's decomposition, where

$$\mathbf{A} = \begin{bmatrix} 1.44 & -0.36 & 5.52 & 0.00 \\ -0.36 & 10.33 & -7.78 & 0.00 \\ 5.52 & -7.78 & 28.40 & 9.00 \\ 0.00 & 0.00 & 9.00 & 61.00 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 0.04 \\ -2.15 \\ 0 \\ 0.88 \end{bmatrix}$$

Also check the solution.

Solution The following program utilizes the functions **choleski** and **choleskiSol.** The solution is checked by computing **Ax**.

```
% Example 2.8 (Choleski decomposition)
A = [1.44  -0.36  5.52  0.00;
      -0.36 10.33 -7.78 0.00;
       5.52 -7.78 28.40 9.00;
       0.00 0.00 9.00 61.00];
b = [0.04 -2.15 0 0.88];
L = choleski(A);
x = choleskiSol(L,b)
Check = A*x          % Verify the result
x =
      3.0921
     -0.7387
     -0.8476
      0.1395
Check =
       0.0400
      -2.1500
      -0.0000
       0.880
```

Lecture 3: Interpolation

Polynomial Interpolation

1. Lagrange's Method

The simplest form of an interpolant is a polynomial. It is always possible to construct a unique polynomial $Pn-1(x)$ of degree $n - 1$ that passes through n distinct data points. One means of obtaining this polynomial is the *formula of Lagrange*

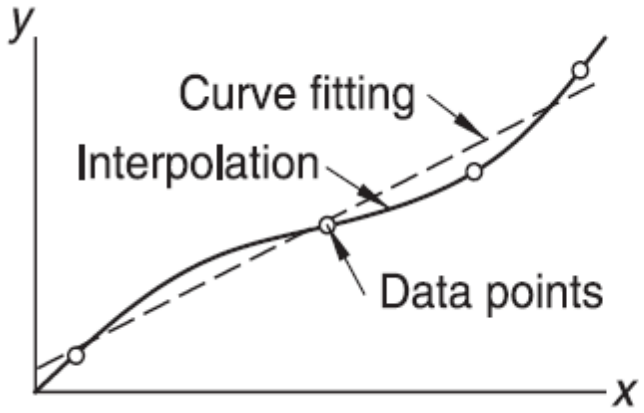


Figure 3.1. Interpolation and curve fitting of data.

$$P_{n-1}(x) = \sum_{i=1}^{n} y_i \ell_i(x) \qquad (3.1a)$$

where

$$\ell_i(x) = \frac{x - x_1}{x_i - x_1} \cdot \frac{x - x_2}{x_i - x_2} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \cdot \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n}$$

$$= \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_i}{x_i - x_j}, \quad l = 1, 2, \ldots, n \qquad (3.1b)$$

are called the *cardinal functions*.

For example, if $n = 2$, the interpolant is the straight line $P_1(x) = y_1 l_1(x) + y_2 l_2(x)$, where

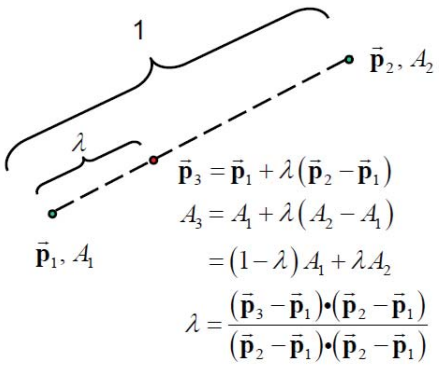
$$\ell_1(x) = \frac{x - x_2}{x_1 - x_2} \qquad \ell_2(x) = \frac{x - x_1}{x_2 - x_1}$$

With $n = 3$, interpolation is parabolic: $P_2(x) = y_1 l_1(x) + y_2 l_2(x) + y_3 l_3(x)$, where now

$$\ell_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}$$

$$\ell_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}$$

$$\ell_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$



The *cardinal functions* are polynomials of degree $n - 1$ and have the property

$$\ell_i(x_j) = \left\{ \begin{array}{l} 0 \text{ if } l \neq j \\ 1 \text{ if } l = j \end{array} \right\} = \delta_{ij} \qquad (3.2)$$

where δ_{ij} is the Kronecker delta. This property is illustrated in Fig. 3.2 for three-point interpolation ($n = 3$) with $x_1 = 0$, $x_2 = 2$, and $x_3 = 3$.

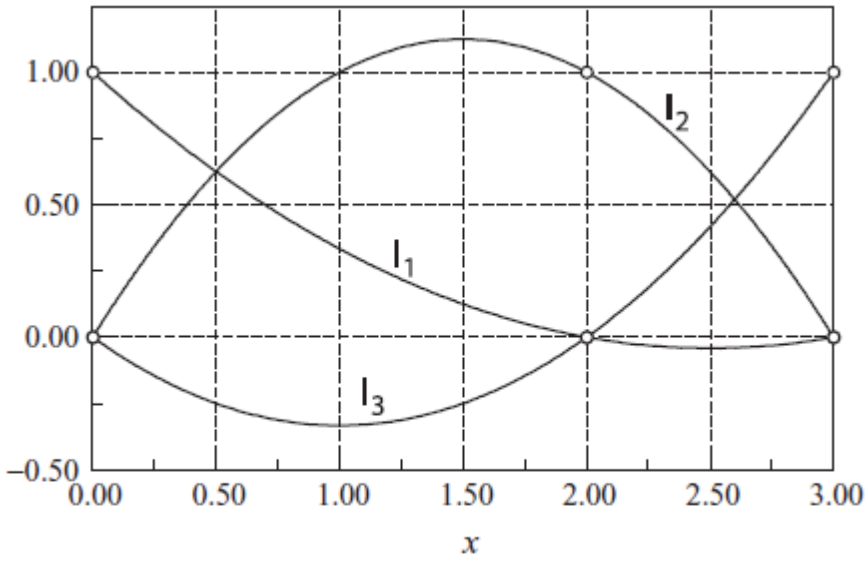


Figure 3.2. Example of quadratic cardinal functions.

To prove that the interpolating polynomial passes through the data points, we substitute $x = xj$ into Eq. (3.1a) and then utilize Eq. (3.2). The result is

$$P_{n-1}(x_j) = \sum_{i=1}^{n} y_i \ell_i(x_j) = \sum_{i=1}^{n} y_i \delta_{ij} = y_j$$

It can be shown that the error in polynomial interpolation is

$$f(x) - P_{n-1}(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{n!} f^{(n)}(\xi) \qquad (3.3)$$

where ξ lies somewhere in the interval ($x1$, xn); its value is otherwise unknown. It is instructive to note that the further data point is from x, the more it contributes to the error at x.

2. Newton's Method

Evaluation of Polynomial

Although Lagrange's method is conceptually simple, it does not lend itself to an efficient algorithm. A better computational procedure is obtained with Newton's method, where the interpolating polynomial is written in the form

$$P_{n-1}(x) = a_1 + (x - x_1)a_2 + (x - x_1)(x - x_2)a_3 + \cdots + (x - x_1)(x - x_2) \cdots (x - x_{n-1})a_n$$

This polynomial lends itself to an efficient evaluation procedure. Consider, for example, four data points ($n = 3$). Here the interpolating polynomial is

$$P_3(x) = a_1 + (x - x_1)a_2 + (x - x_1)(x - x_2)a_3 + (x - x_1)(x - x_2)(x - x_3)a_4$$

$$= a_1 + (x - x_1)\{a_2 + (x - x_2)[a_3 + (x - x_3)a_4]\}$$

which can be evaluated backwards with the following recurrence relations:

$$P_0(x) = a_4$$

$$P_1(x) = a_3 + (x - x_3)P_0(x)$$

$$P_2(x) = a_2 + (x - x_2)P_1(x)$$

$$P_3(x) = a_1 + (x - x_1)P_2(x)$$

For arbitrary n we have

$$P_0(x) = a_n \qquad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, \ldots, n-1 \qquad (3.4)$$

newtonPoly

Denoting the *x*-coordinate array of the data points by xData, and the number of data points by *n*, we have the following algorithm for computing $P_{n-1}(x)$:

```
function p = newtonPoly(a,xData,x)
% Returns value of Newton's polynomial at x.
% USAGE: p = newtonPoly(a,xData,x)
% a           = coefficient array of the polynomial;
%               must be computed first by newtonCoeff.
% xData = x-coordinates of data points.
n = length(xData);
p = a(n);
for k = 1: n-1;
    p = a(n-k) + (x - xData(n-k))*p;
end
```

Computation of Coefficients

The coefficients of $P_{n-1}(x)$ are determined by forcing the polynomial to pass through each data point: $y_i = P_{n-1}(x_i)$, $i = 1, 2, \ldots, n$. This yields the simultaneous equations

$$\begin{aligned} y_1 &= a_1 \\ y_2 &= a_1 + (x_2 - x_1)a_2 \\ y_2 &= a_1 + (x_3 - x_1)a_2 + (x_3 - x_1)(x_3 - x_2)a_3 \qquad \text{(a)} \\ &\vdots \\ y_n &= a_1 + (x_n - x_1)a_1 + \cdots + (x_n - x_1)(x_n - x_2)\cdots(x_n - x_{n-1})a_n \end{aligned}$$

Introducing the *divided differences*

$$\begin{aligned} \nabla y_i &= \frac{y_i - y_1}{x_i - x_1}, \quad i = 2, 3, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_2}{x_i - x_2}, \quad i = 3, 4, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_3}{x_i - x_3}, \quad i = 4, 5, \ldots n \qquad (3.5) \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}} \end{aligned}$$

the solution of Eqs. (a) is

$$a_1 = y_1 \qquad a_2 = \nabla y_2 \qquad a_3 = \nabla^2 y_3 \qquad \cdots \qquad a_n = \nabla^n y_n \qquad (3.6)$$

If the coefficients are computed by hand, it is convenient to work with the format in Table 3.1 (shown for $n = 5$).

x_1	y_1				
x_2	y_2	∇y_2			
x_3	y_3	∇y_3	$\nabla^2 y_3$		
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	
x_5	y_5	∇y_5	$\nabla^2 y_5$	$\nabla^3 y_5$	$\nabla^4 y_5$

Table 3.1

The diagonal terms $(y_1, \nabla y_2, \nabla^2 y_3, \nabla^3 y_4, \text{ and } \nabla^4 y_5)$ in the table are the coefficients of the polynomial. If the data points are listed in a different order, the entries in the table will change, but the resultant polynomial will be the same – recall that a polynomial of degree $n - 1$ interpolating n distinct data points is unique.

newtonCoeff

Machine computations are best carried out within a one-dimensional array **a** employing the following algorithm:

```
function a = newtonCoeff(xData,yData)
% Returns coefficients of Newton's polynomial.
% USAGE: a = newtonCoeff(xData,yData)
% xData = x-coordinates of data points.
% yData = y-coordinates of data points.
n = length(xData);
a = yData;
for k = 2: n
    a(k: n) = (a(k: n) - a(k-1))./(xData(k: n) - xData(k-1));
end
```

Initially, **a** contains the *y* values of the data, so that it is identical to the second column in Table 3.1. Each pass through the for-loop generates the entries in the next column, which overwrite the corresponding elements of **a**. Therefore, **a** ends up containing the diagonal terms of Table 3.1; that is, the coefficients of the polynomial.

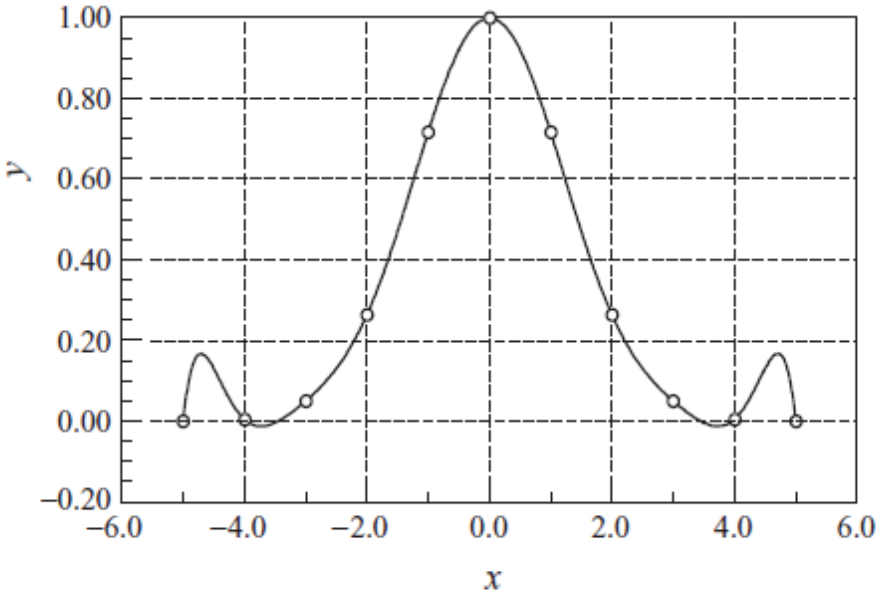


Figure 3.3. Polynomial interpolant displaying oscillations.

Limitations of Polynomial Interpolation

Polynomial interpolation should be carried out with the fewest feasible number of data points. Linear interpolation, using the nearest two points, is often sufficient if the data points are closely spaced. Three to six nearest-neighbor points produce good results in most cases. An interpolant intersecting more than six points must be viewed with suspicion. The reason is that the data points that are far from the point of interest do not contribute to the accuracy of the interpolant. In fact, they can be detrimental.

The danger of using too many points is illustrated in Fig. 3.3. There are 11 equally spaced data points represented by the circles. The solid line is the interpolant, a polynomial of degree ten, that intersects all the points. As seen in the figure, a polynomial of such a high degree has a tendency to oscillate excessively between the data points. A much smoother result would be obtained by using a cubic interpolant spanning
four nearest-neighbor points.

Polynomial extrapolation (interpolating outside the range of data points) is dangerous. As an example, consider Fig. 3.4. There are six data points, shown as circles. The fifth-degree interpolating polynomial is represented by the solid line. The interpolant looks fine within the range of data points, but drastically departs from the obvious trend when $x > 12$. Extrapolating y at $x = 14$, for example, would be absurd in this case.

If extrapolation cannot be avoided, the following three measures can be useful:

- Plot the data and visually verify that the extrapolated value makes sense.

- Use a low-order polynomial based on nearest-neighbor data points. Linear or quadratic interpolant, for example, would yield a reasonable estimate of $y(14)$ for the data in Fig. 3.4.

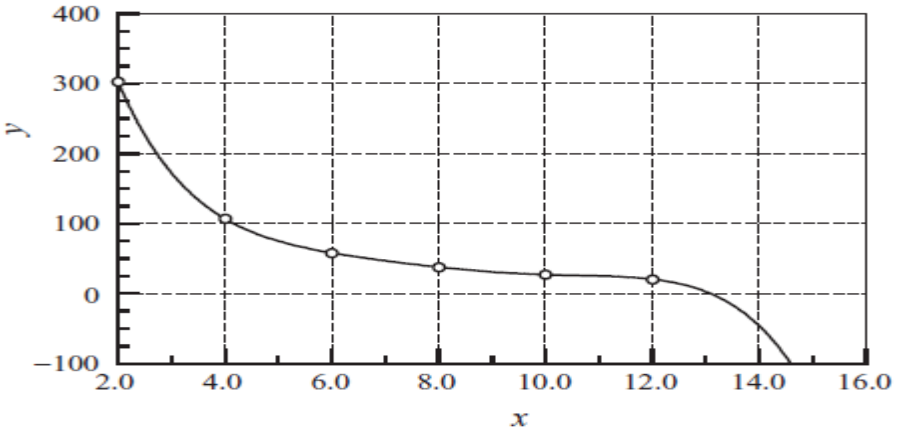


Figure 3.4. Extrapolation may not follow the trend of data.

- Work with a plot of log x versus log y, which is usually much smoother than the x–y curve, and thus safer to extrapolate. Frequently, this plot is almost a straight line. This is illustrated in Fig. 3.5, which represents the logarithmic plot of the data in Fig. 3.4.

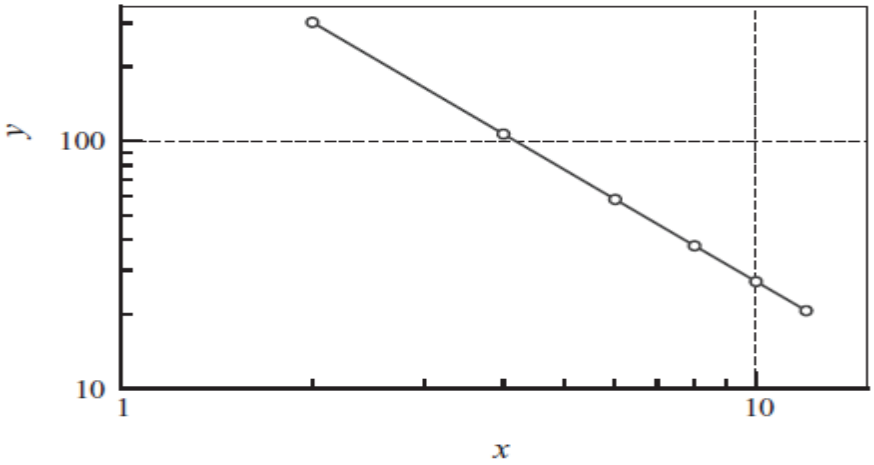


Figure 3.5. Logarithmic plot of the data in Fig. 3.4.

EXAMPLE 3.1

Given the data points

x	0	2	3
y	7	11	28

use Lagrange's method to determine y at $x = 1$.

Solution

$$\ell_1 = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(1-2)(1-3)}{(0-2)(0-3)} = \frac{1}{3}$$

$$\ell_2 = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(1-0)(1-3)}{(2-0)(2-3)} = 1$$

$$\ell_3 = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(1-0)(1-2)}{(3-0)(3-2)} = -\frac{1}{3}$$

$$y = y_1\ell_1 + y_2\ell_2 + y_3\ell_3 = \frac{7}{3} + 11 - \frac{28}{3} = 4$$

EXAMPLE 3.2

The data points

x	−2	1	4	−1	3	−4
y	−1	2	59	4	24	−53

lie on a polynomial. Determine the degree of this polynomial by constructing the divided difference table, similar to Table 3.1.

Solution

i	x_i	y_i	∇y_i	$\nabla^2 y_i$	$\nabla^3 y_i$	$\nabla^4 y_i$	$\nabla^5 y_i$
1	−2	−1					
2	1	2	1				
3	4	59	10	3			
4	−1	4	5	−2	1		
5	3	24	5	2	1	0	
6	−4	−53	26	−5	1	0	0

Here are a few sample calculations used in arriving at the figures in the table:

$$\nabla y_3 = \frac{y_3 - y_1}{x_3 - x_1} = \frac{59 - (-1)}{4 - (-2)} = 10$$

$$\nabla^2 y_3 = \frac{\nabla y_3 - \nabla y_2}{x_3 - x_2} = \frac{10 - 1}{4 - 1} = 3$$

$$\nabla^3 y_6 = \frac{\nabla^2 y_6 - \nabla^2 y_3}{x_6 - x_3} = \frac{-5 - 3}{-4 - 4} = 1$$

From the table, we see that the last nonzero coefficient (last nonzero diagonal term) of Newton's polynomial is $\nabla^3 y_3$, which is the coefficient of the cubic term. Hence the polynomial is a cubic.

EXAMPLE 3.3

The data points in the table lie on the plot of $f(x) = 4.8\cos \pi x/20$. Interpolate this data by Newton's method at $x = 0, 0.5, 1.0, \ldots, 8.0$ and compare the results with the "exact" values given by $y = f(x)$.

x	0.15	2.30	3.15	4.85	6.25	7.95
y	4.79867	4.49013	4.2243	3.47313	2.66674	1.51909

Solution

```
% Example 3.3 (Newton's interpolation)
xData = [0.15; 2.3; 3.15; 4.85; 6.25; 7.95];
yData = [4.79867; 4.49013; 4.22430; 3.47313;...
2.66674; 1.51909];
a = newtonCoeff(xData,yData);
'       x      yInterp      yExact'
for x = 0: 0.5: 8
      y = newtonPoly(a,xData,x);
      yExact = 4.8*cos(pi*x/20);
      fprintf('%10.5f',x,y,yExact)
      fprintf('\n')
end
```

The results are:

ans =

x	yInterp	yExact
0.00000	4.80003	4.80000
0.50000	4.78518	4.78520
1.00000	4.74088	4.74090
1.50000	4.66736	4.66738
2.00000	4.56507	4.56507
2.50000	4.43462	4.43462

x	yInterp	yExact
3.00000	4.27683	4.27683
3.50000	4.09267	4.09267
4.00000	3.88327	3.88328
4.50000	3.64994	3.64995
5.00000	3.39411	3.39411

3. Interpolation with Cubic Spline

If there are more than a few data points, a cubic spline is hard to beat as a global interpolant. It is considerably "stiffer" than a polynomial in the sense that it has less tendency to oscillate between data points.

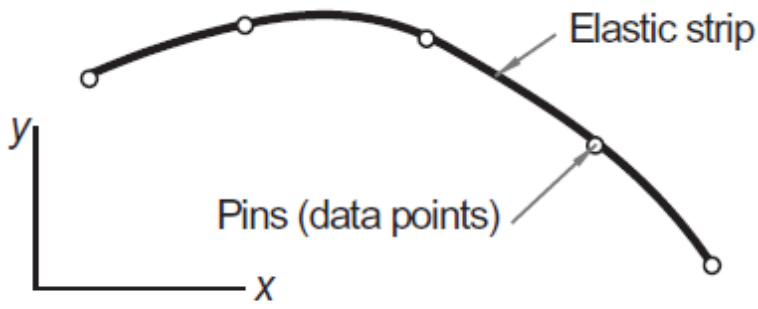


Figure 3.6. Mechanical model of natural cubic spline.

The mechanical model of a cubic spline is shown in Fig. 3.6. It is a thin, elastic strip that is attached with pins to the data points. Because the strip is unloaded between the pins, each segment of the spline curve is a cubic polynomial – recall from beam theory that the differential equation for the displacement of a beam is $d4y/dx4 = q/(EI)$, so that $y(x)$ is a cubic since the load q vanishes. At the pins, the slope and bending moment (and hence the second derivative) are continuous. There is no bending moment at the two end pins; hence, the second derivative of the spline is zero at the end points. Since these end conditions occur naturally in the beam model, the resulting curve is known as the *natural cubic spline*. The pins, that is, the data points are called the *knots* of the spline.

Figure 3.7 shows a cubic spline that spans n knots. We use the notation $fi,i+1(x)$ for the cubic polynomial that spans the segment between knots i and $i + 1$. Note that the spline is a *piecewise cubic* curve, put together from the $n - 1$ cubics $f1,2(x)$, $f2,3(x)$, . . . , $fn-1,n(x)$, all of which have different coefficients.

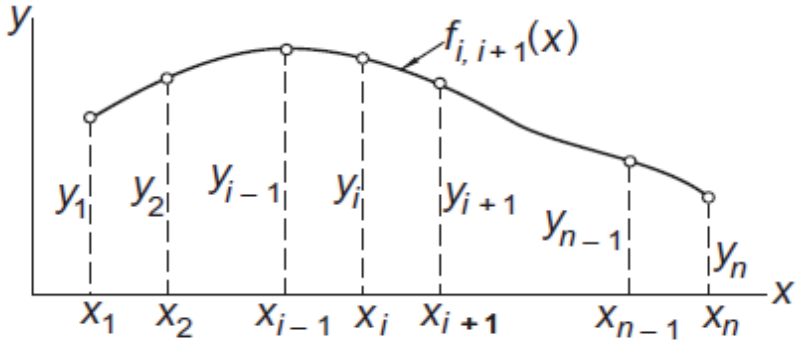


Figure 3.7. Cubic spline.

Denoting the second derivative of the spline at knot i by ki , continuity of second derivatives requires that

$$f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i \qquad \text{(a)}$$

At this stage, each k is unknown, except for

$$k_1 = k_n = 0$$

The starting point for computing the coefficients of $f_{i,i+1}(x)$ is the expression for $f''_{i,i+1}(x)$, which we know to be linear. Using Lagrange's two-point interpolation, we can write

$$f''_{i,i+1}(x) = k_i \ell_i(x) + k_{i+1} \ell_{i+1}(x)$$

where

$$\ell_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} \qquad \ell_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$$

Therefore,

$$f''_{i,i+1}(x) = \frac{k_i(x - x_{i+1}) - k_{i+1}(x - x_i)}{x_i - x_{i+1}} \qquad \text{(b)}$$

Integrating twice with respect to x, we obtain

$$f_{i,i+1}(x) = \frac{k_i(x - x_{i+1})^3 - k_{i+1}(x - x_i)^3}{6(x_i - x_{i+1})} + A(x - x_{i+1}) - B(x - x_i) \qquad \text{(c)}$$

where A and B are constants of integration. The last two terms in Eq. (c) would usually be written as $Cx + D$. By letting $C = A - B$ and $D = -Ax_{i+1} + Bx_i$, we end up with the terms in Eq. (c), which are more convenient to use in computations that follow.

Imposing the condition $f_{i,i+1}(x_i) = y_i$, we get from Eq. (c)

$$\frac{k_i(x_i - x_{i+1})^3}{6(x_i - x_{i+1})} + A(x_i - x_{i+1}) = y_i$$

Therefore,

$$A = \frac{y_i}{x_i - x_{i+1}} - \frac{k_i}{6}(x_i - x_{i+1}) \qquad \text{(d)}$$

Similarly, $f_{i,i+1}(x_{i+1}) = y_{i+1}$ yields

$$B = \frac{y_{i+1}}{x_i - x_{i+1}} - \frac{k_{i+1}}{6}(x_i - x_{i+1}) \qquad \text{(e)}$$

Substituting Eqs. (d) and (e) into Eq. (c) results in

$$\begin{aligned} f_{i,i+1}(x) = {} & \frac{k_i}{6}\left[\frac{(x - x_{i+1})^3}{x_i - x_{i+1}} - (x - x_{i+1})(x_i - x_{i+1})\right] \\ & - \frac{k_{i+1}}{6}\left[\frac{(x - x_i)^3}{x_i - x_{i+1}} - (x - x_i)(x_i - x_{i+1})\right] \\ & + \frac{y_i(x - x_{i+1}) - y_{i+1}(x - x_i)}{x_i - x_{i+1}} \end{aligned} \qquad (3.10)$$

The second derivatives ki of the spline at the interior knots are obtained from the slope continuity conditions $f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i)$, where $i = 2, 3, \ldots, n - 1$. After a little algebra, this results in the simultaneous eqs

$$\begin{aligned} & k_{i-1}(x_{i-1} - x_i) + 2k_i(x_{i-1} - x_{i+1}) + k_{i+1}(x_i - x_{i+1}) \\ & = 6\left(\frac{y_{i-1} - y_i}{x_{i-1} - x_i} - \frac{y_i - y_{i+1}}{x_i - x_{i+1}}\right), \qquad i = 2, 3, \ldots, n - 1 \end{aligned} \qquad (3.11)$$

Because Eqs. (3.11) have a tridiagonal coefficient matrix, they can be solved economically with functions LUdec3 and LUsol3 described in Section 2.4.

If the data points are evenly spaced at intervals h, then $x_{i-1} - x_i = x_i - x_{i+1} = -h$, and the Eqs. (3.11) simplify to

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1}), \quad i = 2, 3, \ldots, n - 1 \qquad (3.12)$$

splineCurv

The first stage of cubic spline interpolation is to set up Eqs. (3.11) and solve them for the unknown ks (recall that $k1 = kn = 0$). This task is carried out by the function **splineCurv:**

```
function k = splineCurv(xData,yData)
% Returns curvatures of a cubic spline at the knots.
% USAGE: k = splineCurv(xData,yData)
% xData = x-coordinates of data points.
% yData = y-coordinates of data points.
n = length(xData);
c = zeros(n-1,1); d = ones(n,1);
e = zeros(n-1,1); k = zeros(n,1);
c(1: n-2) = xData(1: n-2) - xData(2: n-1);
d(2: n-1) = 2*(xData(1: n-2) - xData(3: n));
e(2: n-1) = xData(2: n-1) - xData(3: n);
k(2: n-1) = 6*(yData(1: n-2) - yData(2: n-1))...
            ./(xData(1: n-2) - xData(2: n-1))...
          - 6*(yData(2: n-1) - yData(3: n))...
            ./(xData(2: n-1) - xData(3: n));
[c,d,e] = LUdec3(c,d,e);
k = LUsol3(c,d,e,k);
```

splineEval

The function splineEval computes the interpolant at *x* from Eq. (3.10). The subfunction findSeg finds the segment of the spline that contains *x* by the method of bisection. It returns the segment number; that is, the value of the subscript *i* in Eq. (3.10).

```
function y = splineEval(xData,yData,k,x)
% Returns value of cubic spline interpolant at x.
% USAGE: y = splineEval(xData,yData,k,x)
% xData = x-coordinates of data points.
% yData = y-coordinates of data points.
% k = curvatures of spline at the knots;
% returned by the function splineCurv.
i = findSeg(xData,x);
h = xData(i) - xData(i+1);
y = ((x - xData(i+1))^3/h - (x - xData(i+1))*h)*k(i)/6.0...
    - ((x - xData(i))^3/h - (x - xData(i))*h)*k(i+1)/6.0...
    + yData(i)*(x - xData(i+1))/h...
    - yData(i+1)*(x - xData(i))/h;
function i = findSeg(xData,x)
% Returns index of segment containing x.
iLeft = 1; iRight = length(xData);
while 1
if(iRight - iLeft) <= 1
        i = iLeft; return
end
i = fix((iLeft + iRight)/2);
if x < xData(i)
      iRight = i;
else
      iLeft = i;
end
end
```

EXAMPLE 3.7

Use natural cubic spline to determine *y* at *x* = 1.5. The data points are

x	1	2	3	4	5
y	0	1	0	1	0

Solution The five knots are equally spaced at *h* = 1. Recalling that the second derivative of a natural spline is zero at the first and last knot, we have *k*1 = *k*5 = 0. The second derivatives at the other knots are obtained from Eq. (3.12). Using *i* = 2, 3, 4, we get the simultaneous equations

$$0 + 4k2 + k3 = 6\,[0 - 2(1) + 0] = -12$$
$$k2 + 4k3 + k4 = 6\,[1 - 2(0) + 1] = 12$$
$$k3 + 4k4 + 0 = 6\,[0 - 2(1) + 0] = -12$$

The solution is *k*2 = *k*4 = −30/7, *k*3 = 36/7.

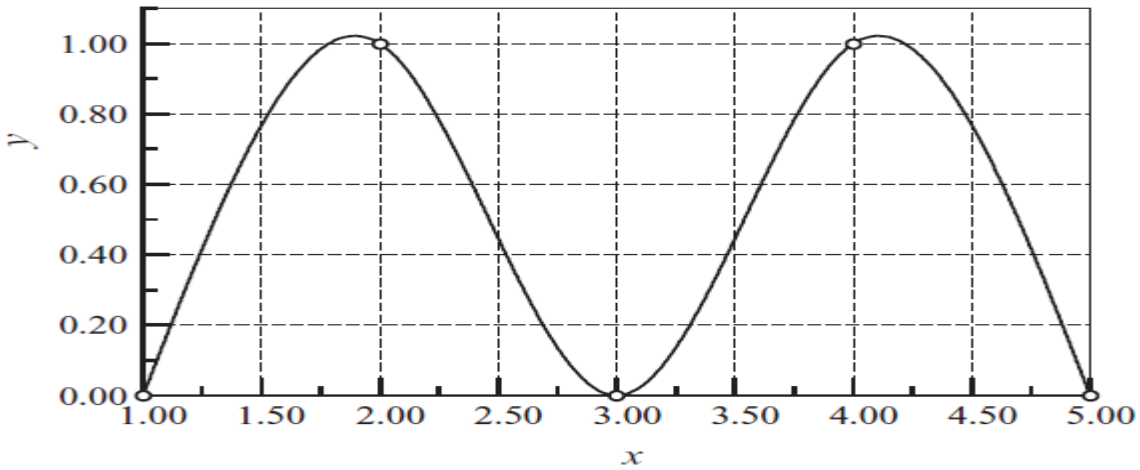
The point *x* = 1.5 lies in the segment between knots 1 and 2. The corresponding interpolant is obtained from Eq. (3.10) by setting *i* = 1.With *xi* − *xi*+1 = −*h*= −1, we obtain

$$f_{1,2}(x) = -\frac{k_1}{6}\left[(x-x_2)^3 - (x-x_2)\right] + \frac{k_2}{6}\left[(x-x_1)^3 - (x-x_1)\right] - \left[y_1(x-x_2) - y_2(x-x_1)\right]$$

Therefore,

$$y(1.5) = f_{1,2}(1.5)$$
$$= 0 + \frac{1}{6}\left(-\frac{30}{7}\right)\left[(1.5-1)^3 - (1.5-1)\right] - \left[0 - 1(1.5-1)\right]$$
$$= 0.7679$$

The plot of the interpolant, which in this case is made up of four cubic segments, is shown in the figure.



EXAMPLE 3.8

Sometimes it is preferable to replace one or both of the end conditions of the cubic spline with something other than the natural conditions. Use the end condition $f'_{1,2}(0) = 0$ (zero slope), rather than $f''_{1,2}(0) = 0$ (zero curvature), to determine the cubic spline interpolant at $x = 2.6$ based on the data points

x	0	1	2	3
y	1	1	0.5	0

Solution We must first modify Eqs. (3.12) to account for the new end condition. Setting $i = 1$ in Eq. (3.10) and differentiating, we get

$$f'_{1,2}(x) = \frac{k_1}{6}\left[3\frac{(x-x_2)^2}{x_1-x_2} - (x_1-x_2)\right] - \frac{k_2}{6}\left[3\frac{(x-x_1)^2}{x_1-x_2} - (x_1-x_2)\right] + \frac{y_1-y_2}{x_1-x_2}$$

Thus the end condition $f'_{1,2}(x_1) = 0$ yields

$$\frac{k_1}{3}(x_1-x_2) + \frac{k_2}{6}(x_1-x_2) + \frac{y_1-y_2}{x_1-x_2} = 0$$

or

$$2k_1 + k_2 = -6\frac{y_1-y_2}{(x_1-x_2)^2}$$

From the given data, we see that $y_1 = y_2 = 1$, so that the last equation becomes

$$2k_1 + k_2 = 0 \qquad \text{(a)}$$

The other equations in Eq. (3.12) are unchanged. Noting that $k_4 = 0$ and $h = 1$, they are

$$k_1 + 4k_2 + k_3 = 6\,[1 - 2(1) + 0.5] = -3 \qquad \text{(b)}$$

$$k_2 + 4k_3 = 6\,[1 - 2(0.5) + 0] = 0 \qquad \text{(c)}$$

The solution of Eqs. (a)–(c) is $k_1 = 0.4615$, $k_2 = -0.9231$, $k_3 = 0.2308$.

The interpolant can now be evaluated from Eq. (3.10). Substituting $i = 3$ and $x_i - x_{i+1} = -1$, we obtain

$$f_{3,4}(x) = \frac{k_3}{6}\left[-(x-x_4)^3 + (x-x_4)\right] - \frac{k_4}{6}\left[-(x-x_3)^3 + (x-x_3)\right] - y_3(x-x_3) + y_4(x-x_2)$$

Therefore,

$$y(2.6) = f_{3,4}(2.6) = \frac{0.2308}{6}\left[-(-0.4)^3 + (-0.4)\right] + 0 - 0.5(-0.4) + 0$$

$$= 0.1871$$

EXAMPLE 3.9

Write a program that interpolates between given data points with the natural cubic spline. The program must be able to evaluate the interpolant for more than one value of *x*. As a test, use data points specified in Example 3.7 and compute the interpolant at *x* = 1.5 and *x* = 4.5 (due to symmetry, these values should be equal).

Solution The program below prompts for *x*; it is terminated by pressing the "return" key.

```
% Example 3.9 (Cubic spline)
xData = [1; 2; 3; 4; 5];
yData = [0; 1; 0; 1; 0];
k = splineCurv(xData,yData);
while 1
    x = input('x = ');
    if isempty(x)
        fprintf('Done'); break
    end
    y = splineEval(xData,yData,k,x)
    fprintf('\n')
end
```

Running the program produces the following results:

```
x = 1.5
y = 0.7679
x = 4.5
y = 0.7679
x =
Done
```

4. Two Dimension Interpolation

Now suppose you have some values of a function z = f(x, y).

• For example: a digital elevation map: you want to generate nice smooth contours for hiking maps based on grids of elevation as a function of latitude and longitude that can be downloaded from USGS. This is going to require computing values at points that are between the grid locations where you are given elevation.

• For example: Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) medical data. These scanners produce maps of density (or other physical properties) as a function of 3D position, generally for distinct slices through a body. We may need to shift and rotate one slice relative to the next, due to motion of the patient between scans. Without interpolation we are limited to shifting by an integer multiple of the grid spacing, but with the ability to evaluate between the points, we can shift it exactly as much as is needed.

There are two distinct generalizations of linear interpolation to 2D- bilinear and what I'll call "truly linear."

4.1.a) Lagrange in 2D interpolation

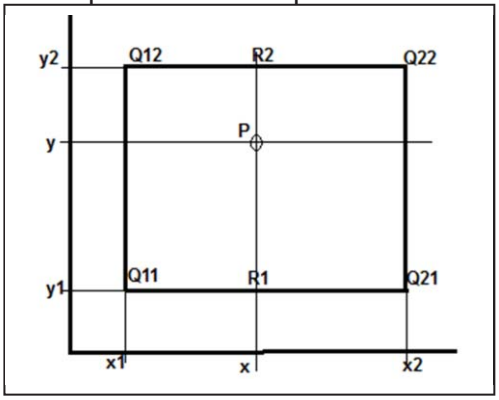
In mathematics, the extension of Linear Interpolation is called Bilinear Interpolation. It was used for interpolating functions of two variables on a regular grid. The key idea is to perform L. interpolation first in one direction, and then in the other direction .

Suppose that we want to find the value of the unknown f at the point *P(x,y)*, as in Fig.. It is assumed that we know the value of f at the four points Q11= (x1, y1) , Q12= (x1, y2)

Q21= (x2, y1) , Q22= (x2, y2);

We first do Linear Interpolation in the x-direction. This yields

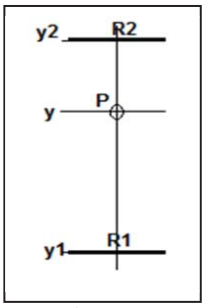
$$f(R1) \approx \frac{x2 - x}{x2 - x1} f(Q11) + \frac{x - x1}{x2 - x1} f(Q21), \ where \ R1 = (x, y1)$$



By using Linear Interpolation

$$f1(x) = \frac{x-x1}{x0-x1} f(x0) + \frac{x-x0}{x1-x0} f(x1); \quad Also;$$

$$f(R2) \approx \frac{x2-x}{x2-x1} f(Q12) + \frac{x-x1}{x2-x1} f(Q22), \quad where \ R2 = (x, y2)$$



We proceed by interpolating in y- direction

$$f(p) \approx \frac{y2-y}{y2-y1} f(R1) + \frac{y-y1}{y2-y1} f(R2)$$

This gives us the desired estimate of *f(x,y)*.

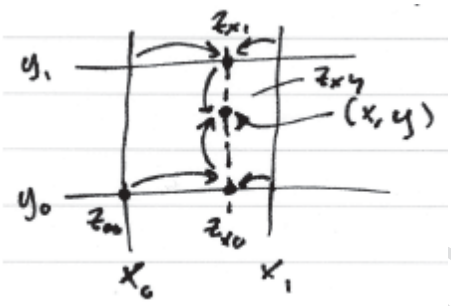
$$f(x,y) \approx \frac{y2-y}{y2-y1}\left[\frac{x2-x}{x2-x1} f(Q11) + \frac{x-x1}{x2-x1} f(Q21)\right] + \frac{y-y1}{y2-y1}\left[\frac{x2-x}{x2-x1} f(Q12) + \frac{x-x1}{x2-x1} f(Q22)\right]$$

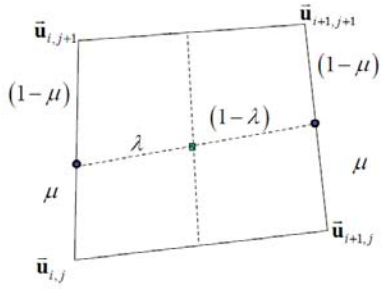
$\rightarrow \rightarrow$

$$f(x,y) = \left[\frac{(x2-x)}{(x2-x1)}\frac{(y2-y)}{(y2-y1)} f(Q11) + \frac{(x-x1)}{(x2-x1)}\frac{(y2-y)}{(y2-y1)} f(Q21)\right] + \left[\frac{(x2-x)}{(x2-x1)}\frac{(y-y1)}{(y2-y1)} f(Q12) + \frac{(x-x1)}{(x2-x1)}\frac{(y-y1)}{(y2-y1)} f(Q22)\right]$$

4.1 .b) Bilinear interpolation

Bilinear interpolation is by far the more common. The idea is to interpolate along one dimension using values that were themselves interpolated along the other dimension.





$$\vec{u}(\lambda,\mu) = \lambda\left(\mu\vec{u}_{i+1,j+1} + (1-\mu)\vec{u}_{i+1,j}\right) + (1-\lambda)\left(\mu\vec{u}_{i,j+1} + (1-\mu)\vec{u}_{i,j}\right)$$
$$= \vec{u}_{i,j} + \lambda\left(\vec{u}_{i+1,j} - \vec{u}_{i,j}\right) + \mu\left(\vec{u}_{i,j+1} - \vec{u}_{i,j}\right) + \lambda\mu\left(\vec{u}_{i+1,j+1} - \vec{u}_{i,j}\right)$$

(Here I have renamed the points x_0, x_1 rather than x_k, x_{k+1} to keep the indices from getting out of hand.)

If I had values at (x, y_0) and (x, y_1) then I could linearly interpolate along the vertical line. No problem just generate them by interpolating along the horizontals!

$$z_{x0} = (1-\alpha)z_{00} + \alpha\, z_{10} \qquad \alpha = (x - x_0)/(x_1 - x_0)$$
$$z_{x1} = (1-\alpha)z_{01} + \alpha\, z_{11}$$
$$z_{xy} = (1-\beta)z_{x0} + \beta\, z_{x1} \qquad \beta = (y - y_0)/(y_1 - y_0)$$

Note that it does not matter whether I interpolate across and then down or down and then across (i.e. on x first or y first). Either way I end up with

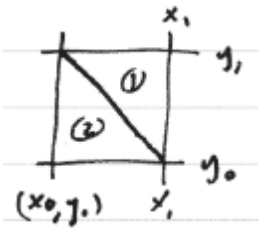
$$z_{xy} = (1-\alpha)(1-\beta)z_{00} + (1-\alpha)\beta z_{01} + \alpha(1-\beta)z_{10} + \alpha\beta z_{11}$$

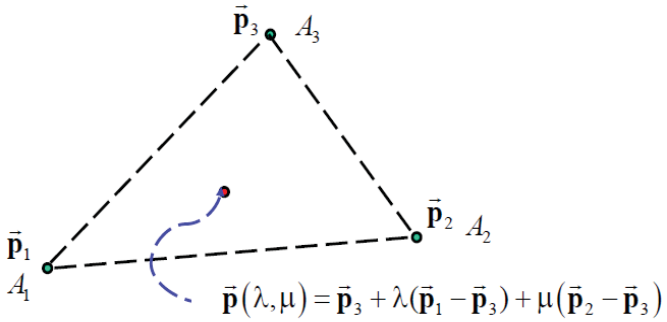
This is bilinear interpolation. It results in a piecewise function that is not piecewise linear of course it can't be, because it matches the data at four different points, and three points uniquely determine the linear function. It has a piece for each cell in the grid of data points, but the interpolation defined over that rectangle is not linear. Look at this most recent equation, remembering that α is a linear function of x and β is a linear function of y. The full expression for z_{xy} is going to contain a constant term, an x term, a y term, and an xy term. Because of the presence of this last term is not linear.

This kind of function is called bilinear because it is linear as a function of x when y is held fixed and also linear as a function of y when x is held fixed.

4.2 **Truly linear interpolation**

However, if we cut the squares along their diagonals and define a linear interpolant for each half, we can make it work:





$$\vec{p}(\lambda,\mu) = \vec{p}_3 + \lambda(\vec{p}_1 - \vec{p}_3) + \mu(\vec{p}_2 - \vec{p}_3)$$
$$= \lambda\vec{p}_1 + \mu\vec{p}_2 + (1-\lambda-\mu)\vec{p}_3$$
$$\vec{p}(\lambda,\mu,\nu) = \lambda\vec{p}_1 + \mu\vec{p}_2 + \nu\vec{p}_3 \quad \text{where } \lambda +$$
$$A(\lambda,\mu,\nu) = \lambda A_1 + \mu A_2 + \nu A_3$$

Where λ+ μ+ ν= 1

- Area (1) has a linear function defined by $(x_1, y_0) \rightarrow z_{10}$, $(x_1, y_1) \rightarrow z_{11}$, and $(x_0, y_1) \rightarrow z_{01}$.
- Area (2) has a linear function defined by $(x_0, y_0) \rightarrow z_{00}$, $(x_1, y_0) \rightarrow z_{10}$, and $(x_0, y_1) \rightarrow z_{01}$.

Of course this will work for any set of triangles, not only for triangles that are made by cutting the squares of a grid in half.

When your data naturally comes with a triangulation, this approach is generally best. When the grid is naturally rectilinear, bilinear is probably most convenient. When there is no organization at all, you can build a triangulation .

4.3 **Transformation into FE characteristics**

There are two shapes of elements used in two-dimensional analysis: the triangle and the quadrilateral. The two basic element shapes may be linear elements or quadratic elements, where linear and quadratic refer to the order of the assumed polynomial displacement interpolation function used within the element area. The linear triangle is the simplest and was the first two-dimensional element developed. Analysts do not use it much now because it requires many more elements to produce a converged and accurate solution compared with the quadrilateral. However, we still examine its formulation both for academic purposes and for its occasional use in coarse to fine mesh transitions for refining models.

The triangular element illustrated in Figure 4-1 defines an area bounded by the three sides connecting three node points. Within the element area the displacement function is assumed to be of the form in equation (4.1),

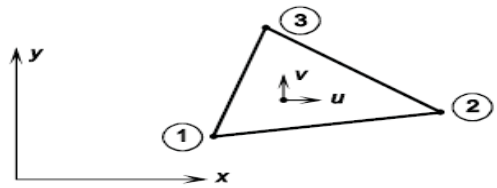


Figure 4-1. Triangular Two-Dimensional Finite Element

$$u = a_1 + a_2 x + a_3 y$$
$$v = a_4 + a_5 x + a_6 y \qquad (4.1)$$

where u and v are displacement components of a material point within the element field, x and y are coordinates of the point, and a_i, i=1,2,...6, are constant coefficients to be determined. This is a linear distribution of the two displacement components for any material point within the element area. The linear function has three undetermined coefficients for each component, and since we have three nodes we may evaluate the three constants by use of the node point values of each component.

Application of the strain-displacement equations to the expressions for *u* and *v* illustrates that all three strain components are constant within the element for this assumed displacement field as derived in equation (4.2).

$$\epsilon_x = \frac{\partial u}{\partial x} = a_2$$

$$\epsilon_y = \frac{\partial v}{\partial y} = a_6 \qquad (4.2)$$

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = a_3 + a_5$$

Also, for homogeneous material throughout the element, the stress-strain relations are all constant; therefore, the stress components are also constant.

This displacement formulation also satisfies the compatibility requirements in the theory of elasticity [4.1] for the continuum. The compatibility requirements are that no gaps or overlaps of material may occur during the process of deformation under load. From these equations, we can see that the continuous nature of the function enforces compatibility within the element. On a triangular element, since the interpolation is linear, any edge formed by connecting two nodes that is a straight line before deformation will remain a straight line after deformation. Therefore, any connecting element using the same two nodes for its shared edge satisfies compatibility.

Some of the early finite element programs [4.2] used the triangle element to create a quadrilateral element by subdividing a quadrilateral shape into four triangles using the centroid of the quadrilateral as their apex. After finding the stiffness matrix for each triangle element, assembly of the triangles and condensation of the internal node resulted in the stiffness matrix of the quadrilateral element. This was an effective way to use the triangular element formulation and employ many more elements without tedious input. However, the element of choice now is an isoparametric quadrilateral formulation.

Next we examine the displacement basis for formulation of the isoparametric quadrilateral element. Taig [4.3] developed the element, and Irons [4.4] published its formulation. The quadrilateral element formulation derives from the formulation of a square element. It uses a co-ordinate system transformation to convert the square to a quadrilateral. Begin with the square element shown in Figure 4-2 with corner nodes.

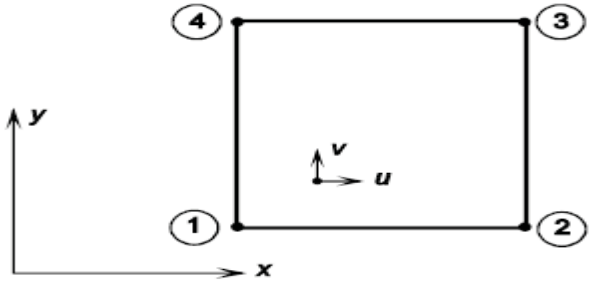


Figure 4-2. Square Two-Dimensional Finite Element

Recognizing that four constants can be evaluated with four nodes, a logical expression for the displacement function components becomes

$$u = a_1 + a_2 x + a_3 y + a_4 xy$$

$$v = a_5 + a_6 x + a_7 y + a_8 xy \; . \qquad (4.3)$$

Use of the strain-displacement relations here shows that

$$\epsilon_x = a_2 + a_4 y$$

$$\epsilon_y = a_7 + a_8 x \qquad (4.4)$$

$$\gamma_{xy} = a_3 + a_4 x + a_6 + a_8 y \; .$$